WL-TR-93-1094

ADA EMBEDDED COMPUTER SOFTWARE SUPPORT (AECSS)

PATRICK ROGERS

SBS ENGINEERING, INC.
18333 EGRET BAY BLVD. SUITE #340
HOUSTON, TX 77058

MARCH 1993

INTERIM REPORT FOR 01/01/90 - 03/01/93

Approved for public release; distribution is unlimited.

AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT PATTERSON AFB OH 45433-7409

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

Kenneth Littlejohn
Project Engineer
WL/AAAF-3

Robert L. Harris
ROBERT L. HARRIS
Chief, Software Concepts Section

Charles H Krueger
CHARLES H. KRUEGER, Chief
Systems Avionics Division
Avionics Directorate

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WL/AAAF-3, WPAFB, OH 45433-7301 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Mar 1993	Interim, 01/01/90--03/01/93

4. TITLE AND SUBTITLE
Ada Embedded Computer Software Support (AECSS)

5. FUNDING NUMBERS
C F33615-89-C-1076
PE 78012
PR 3090
TA 01
WU 09

6. AUTHOR(S)
Patrick Rogers

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SBS Engineering, Inc.
18333 Egret Bay Blvd. #340
Houston TX 77058

8. PERFORMING ORGANIZATION REPORT NUMBER
SBS-AECSS-01I

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Avionics Directorate
Wright Laboratory
Air Force Materiel Command
Wright Patterson AFB OH 45433-7409

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
WL-TR-93-1094

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for Public Release; Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This document describes an implementation of the Ada language which allows applications to be distributed across several processors and chassis, while allowing the application designer to remain within the definition of the language. Application designers use a design method which incorporates restrictions compatible with execution in a distributed target environment. Full rendezvous semantics are supported, including user-defined parameter types and exception propagation. Significant to the effort is the fact that a standard, commercial compilation system was used without modification. Similarly, a standard commercial runtime system was used and augmented to support distributed execution. All code is written in the Ada language itself, including the runtime extensions and the host-based tools.

14. SUBJECT TERMS
Ada, Real-Time, Distribution, Embedded

15. NUMBER OF PAGES
33

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

Table of Contents

Executive Summary

The performance and fault tolerance requirements, as well as the complexity and criticality, of the avionics applications of tomorrow require advances in software production and support that can only be met by distributed, multiprocessing technology. Current computing capabilities do not typically support distribution of any kind -- at best, support is rare, unique, and nonstandard, including that for the standard DoD mission-critical programming language Ada. Although the Ada language directly supports multiprocessing very well, it does not directly support distribution, though some rare implementations do support distribution to some degree.

In 1989, the United States Air Force (WL/AAAF) awarded the Ada Embedded Computer Software Support (AECSS) contract to SBS Engineering, Incorporated. The objective of the AECSS effort is to enhance the support of Ada software for advanced avionics systems. Specifically, the AECSS effort is designed to identify and develop technology to support distributed multiprocessor Ada software. As a result, SBS has implemented system software and tools which allow an Ada applications developer to create unmodified programs that can be distributed across one or more processors in one or more chassis. The language compilation system used is a standard commercial Ada cross-compiler, with extensions by SBS to the runtime system software that executes with the application code. Applications designers work within the standard Ada language, using a design method which incorporates design restrictions compatible with execution on a distributed target environment. For example, full tasking rendezvous are supported, allowing remote communication and synchronization among remote processors. In contrast, shared variables are not allowed in a unit that is to be partitioned among separate processors. A completely "legal" Ada program is nevertheless constructed. Furthermore, within an "atomic" unit (i.e., one that will not be partitioned), no restrictions exist so that the applications designer can use any part of the language.

Central to the implementation is the concept of an *extended* runtime system which executes in support of the application code, performing such activities as memory management, interrupt management, and processor allocation/deallocation. The commercial runtime system defined to support the standard Ada language has been extended by SBS personnel to include support for interactions between tasks on separate processors (possibly in separate chassis). To this end, the source code for the runtime was obtained and altered in order to interact with the extensions. Alterations to the runtime system were very minimal, and occurred in only one file. The runtime extensions themselves are written entirely in the Ada language.

As an AECSS contract extension, SBS will incorporate the distribution technology into an existing avionics application. The results of the technology insertion will be reported in a separate report.

1 Scope

This document describes development performed during the first three years of the Ada Embedded Computer Software Support (AECSS) project, which has been extended to a total of five years. The original three-year project, herein called Phase I, was extended in January 1993. Phase II is intended to incorporate the distribution technology described below into a representative avionics application, which will be reported in a separate report.

2 Purpose

This Project Interim Report describes an implementation of Distributed Ada developed for the AECSS Project. As such, it contains technical material describing the implementation, as well as lessons learned from the experience and other related information.

3 Related Documents

ANSI/MIL-STD-1815A Ada Language Reference Manual, 1983

AECSS Distributed Ada User's Guide, SBS Engineering, 1993

4 Introduction

In 1989, the United States Air Force (WL/AAAF) awarded the AECSS contract to SBS Engineering, Incorporated. The objective of the AECSS effort is to enhance the support of Ada software for advanced avionics systems. Specifically, the AECSS effort is designed to identify and develop technology to support distributed multiprocessor Ada software.

The testbed resulting from this research is composed of homogeneous target processors dispersed over three chassis. Applications developers can create Ada programs in terms of distributable units composed from standard Ada constructs. These units execute in one or more of the processors in one or more of the chassis, and can communicate via standard local or remote rendezvous entry calls. The standard unconditional, conditional and timed entry calls, with both predefined and user-defined parameter types, are supported. Additionally, the required semantics for propagating unhandled exceptions from (remote) accept statements are also supported, again per the language standard.

4.1 Host Hardware

The host is a VaxStation® 3520, executing VAX/VMS®. Ethernet connections exist between the target and host computers. One set of Ethernet connections is available for applications-level communications. The other Ethernet hardware is used solely for downloading and debugging. An X Terminal is also connected to the system via Ethernet connection, and is used as another workstation. Figure 1 shows the host-target interconnection configuration.

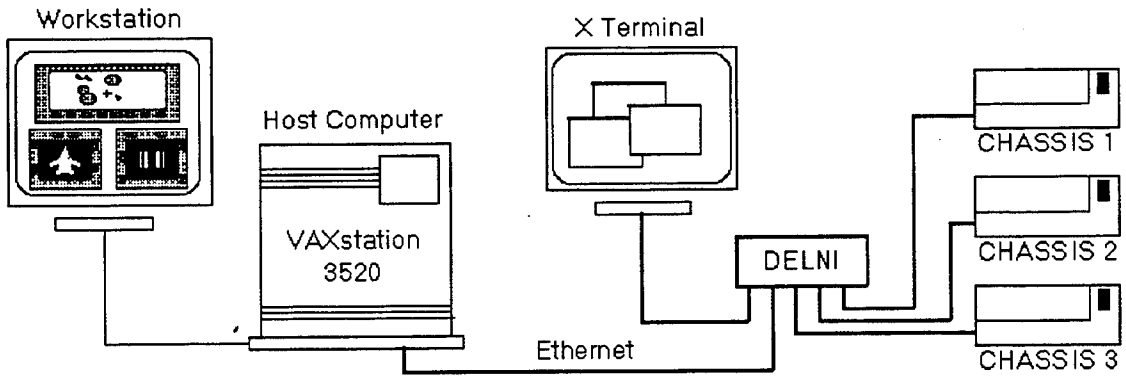

Figure 1. Host - Target Configuration

4.2 Host Software

A standard Ada uniprocessing cross-compilation system, including cross-compiler, linker, downloader and debugger is available from Systems Designers Software, Incorporated. A self-hosted VAX/VMS Ada compiler is available from Digital Equipment Corporation for executing Ada programs on the VAX host. Other development tools, such a configuration management tool and language-sensitive editor are also available.

4.3 Target Hardware

The target execution environment resembles a "typical" avionics platform, in which multiple distributed processors are linked via a high-speed communications network. Each chassis in the AECSS target environment has 8Mb of VME memory, which is available to all processors within the chassis. Each chassis contains at least two commercial-off-the-shelf (COTS) 68030 processors built by FORCE Computer Corporation which have additional hardware functionality supporting multiprocessing. Finally, each chassis is connected to 256Kb of memory called "reflective memory" that is available to all other chassis, forming a common block of global interchassis memory. This global memory was originally to be used by a host computer (executing avionics simulation models) for communicating with the various target computers described above. Soon after the development of the rendezvous within any single chassis, called "intrachassis" rendezvous, Air Force personnel directed SBS to use the global memory to implement the interchassis rendezvous since it would be quickly implementable using an adaptation of the existing intrachassis approach.

4.4 Target Software

SBS personnel retargeted the SD Scicon Ada target kernel to work on the FORCE processors, since the standard product (Motorola) was not selected due to a lack of multiprocessor support.

Application software executing on the target processor(s) appears as a single program to the developer, but is in fact implemented by the underlying support software and tools as a series of one or more distinct interacting programs. Multiprogramming is required of the implementation since the compiler, linker, downloader and debugger are COTS uniprogramming tools. It must be emphasized, however, that the applications developer works in terms of standard Ada.

The source for the runtime systems was purchased, but source for the compiler and linker was not. As a result the distribution support has been designed to work within the constraints of what a validated Ada compiler and linker will accept. For example, the linker must believe that all bodies are present, even for those units that are distributed. This aspect, and the distribution support software architecture in general, will be discussed in detail in later sections.

No mass-media storage devices (e.g., disks) are available on the target system, so there is no file system. Use of package Text_IO is essentially limited to performing I/O to or from a terminal connected to a processor's RS-232 front-panel connector.

5 Research Efforts

5.1 Distributed Ada

The AECSS project is tasked with the development of an implementation of Distributed Ada which follows the language standard as closely as possible. However, because the language does not explicitly support distribution, and in fact has areas that are problematic for distribution support, project personnel have elected to incorporate both reasonable restrictions and current features of the next revision of the language (called "Ada9X"). Issues leading to such decisions, the approaches taken by the project design team, the architecture, and resulting limitations are discussed in the following sections.

5.1.1 Language Issues

The basic execution model, the "Ada virtual machine" implied by the Ada language, is that of a tightly-coupled multiprocessor accessing shared memory [Knight], [Volz], [Rogers]. In particular, the existence of memory that is visible to several units, access values passed as parameters, the definition of a program, library unit elaboration synchronization, a single definition of machine types and a common perception of time all indicate a single homogeneous machine model. Any approach to distribution must address these problems either by attempting to provide support for the model as it exists, or by making changes to the model. These changes may be visible to the application or not, depending on the approach taken. The AECSS approach will be discussed in the following sections.

5.1.2 Partitioning Approaches

In the literature, two major approaches to application-level distribution have emerged [Cornhill], [Knight]. These address the question of how a given set of Ada software units are to be partitioned into executable pieces across a distributed target.

In one approach, called "postpartitioning," the software is partitioned after design and implementation. The application is not designed with distribution explicitly in mind. Rather, distribution is considered a "nonfunctional" aspect of the system, in the same sense that performance and fault-tolerance can be considered characteristics instead of functionality. Postpartitioning results in considerable flexibility, with respect to deployment, since the software does not explicitly contain specific partitioning choices. Different deployment alternatives can be examined until the optimum arrangement is found, in terms of such factors as communications delays, overall throughput, and fault-tolerance. Furthermore, the designer has no more difficult a task than is ordinarily the case since distribution is not an issue during design.

However, this deployment flexibility and applications-level simplicity result in considerable complexity at the systems software level. The runtime system must be prepared to support full language semantics in arbitrary deployment combinations. For instance, if unlimited distribution of entities is supported, individual variables in expressions could be on separate machines, resulting in significant overhead. Postpartitioning implementations typically limit the level of granularity of distributable entity for this reason [Eisenhauer]. Furthermore, since distribution is not considered, the potential for producing a design that does not lead to an optimum partitioning exists.

In the other major approach, termed "prepartitioning," distribution is an explicit design criterion such that the partitioning choices are reflected directly in the software. The design is thus expressed in terms of whatever unit of distribution is supported. Although the designer must explicitly consider distribution, and is thus faced with a more complex task, the likelihood of producing a nonoptimum configuration is significantly diminished because distribution issues are considered as the design progresses. Equally important is the fact that the underlying implementation can be significantly more simple since the "partitions" are conceptually in terms of program units, be they procedures, modules, et cetera, rather than, in the extreme case, arbitrary program variables.

5.1.3 Virtual Nodes

The Virtual Node (VN) approach [Atkinson] describes a combination of prepartitioning and a design methodology in which the resulting design consists of distributable units that conform to methodology-specific restrictions. These restrictions exist to preclude the extensive overhead associated with unlimited use of troublesome language constructs, such as variables visible to software on two different physical nodes. As such the VN method defines only restrictions concerning *internode* connections; no restrictions are placed on the implementation of individual virtual nodes since they are indivisible. Full Ada may therefore be used internally, subject to any restrictions resulting from the nature of the application.

The complexity of the underlying runtime system software, under either partitioning approach described above, depends in part on the unit of distribution chosen. For example, the choice of program units as the level of distribution considerably reduces complexity in comparison to supporting arbitrary distribution of program variables. (Multiprogramming, in fact, can be considered the least complex since it represents the highest level of granularity.) The VN design method, as described in [Atkinson], is based on coarse-granularity units of distribution to reduce this complexity.

5.1.3.1 Virtual Nodes Design Method

As defined by the VN methodology, a virtual node can only communicate with other virtual nodes by remote procedure calls (RPC) or remote entry calls, using subprogram declarations or "interface task" declarations exported from "interface packages". Each VN has one or more such interface packages. Communication via directly referenced shared variables is prohibited as a design choice, as is indirect communication via shared packages that encapsulate state.

With respect to the interface tasks in interface packages, both task abort and references to task attributes are prohibited. Allowing interface task aborts would seriously undermine the running system and is of questionable use in a distributed context. Task attributes are prohibited since interface tasks are the only tasks that could be remotely referenced and they should be always callable and never terminated.

The Virtual Node design method defines the concept of an "Ada node," which represents the unit of deployment for execution. This approach avoids the issues of library unit elaboration, program library closure, and the conceptual problem of a single "main" program. Ada nodes represent the execution entry point of a collection of virtual nodes on a given physical node, without the additional semantics of the "main" subprogram.

The effect of the design method restrictions and partitioning approach is that a legal Ada program is created, which can be tested as a single program on a nondistributed machine (assuming the existence of no other target dependencies). The application designer uses standard Ada without the need of special communications or synchronization facilities. The fact that the underlying implementation is essentially multiprogramming is hidden from the user.

5.1.3.2 AECSS Tailoring

The AECSS approach to distribution of Ada software is based on an adaptation of the Virtual Nodes approach described in [Atkinson] regarding the "DIstributed Ada DEMonstrated" (DIADEM) project, and [Wellings] regarding the "York Distributed Ada" (YDA) project. This adaptation involves both expected implementation choices as well as some variations in anticipation of Ada9X changes.

For example, either RPC or remote rendezvous (or both) can be supported as the communications mechanism between virtual nodes. As was chosen in DIADEM, remote rendezvous is the only mechanism supported by AECSS, although for different reasons. DIADEM chose remote rendezvous for the sake of expressive power, which is a sufficient reason in itself. AECSS chose remote rendezvous for the additional reason of adherence to the language standard. (Note that RPC will be provided by Ada9X compilers that support distribution.)

Unlike DIADEM, the AECSS application source code is not transformed. Instead there exists at most one (potentially empty) "Ada node," which is the "main subprogram" in the normal sense. All other Ada nodes are null procedures that import all mapped virtual nodes for a given physical node; they do not have a source code representation that is visible to the application.

Furthermore, instead of removing the concept of a "main" subprogram, certain language requirements have been relaxed in anticipation of Ada9X changes. In general, required semantics that conflict with "intuitive" distributed execution have been deferred in anticipation of Ada9X. AECSS "Ada nodes" are, in fact, very much like the current Ada9X concept of "partitions," which redefine the concept of an Ada program to a collection of independent, separately elaborated software elements [Intermetrics].

In AECSS, virtual nodes are not represented as procedures to be called from tasks. Virtual nodes are instead collections of one or more application packages, with one or more interface packages per virtual node collection. Additionally, interface packages are allowed to contain more than just interface tasks; in particular, they can contain type declarations.

Unlike DIADEM, AECSS interface tasks are not allowed to have access values as parameters [Atkinson]. This is a partial result of not transforming the code -- in this case, not transforming access objects into {node_id,access_value} pairs.

Finally, AECSS doesn't have a concept of virtual node *types* [Atkinson], since the AECSS implementation is intended to remain as close to standard Ada as reasonably possible and since virtual nodes are represented differently, as described above.

5.1.4 Desired Characteristics

Throughout the design and implementation of the system support software the characteristics of predictability, robustness, and simplicity have been central concerns. For example, with respect

to predictability, stubs are dedicated to their corresponding remote entity rather than being created and/or allocated when needed. The memory allocations for passing parameters between processors in the same chassis, and between processors in different chassis, is completely preallocated and is of fixed size. A predefined, static number of stubs may exist. As a result, there is little initialization time and the timing in general is as predictable as it would be without distribution being involved.

Furthermore, the design is such that local rendezvous incur minimal additional overhead from the remote support. When any rendezvous call takes place, three additional instructions are executed to determine if the called task is remote. (The three are move, test and branch instructions.)

5.1.5 Architecture

The primary details of the implementation architecture, namely, the shared memories, stubs and surrogates, intrachassis and interchassis rendezvous, and timed/conditional entry call design issues are discussed in the following sections. Figure 2 shows the basic architecture.

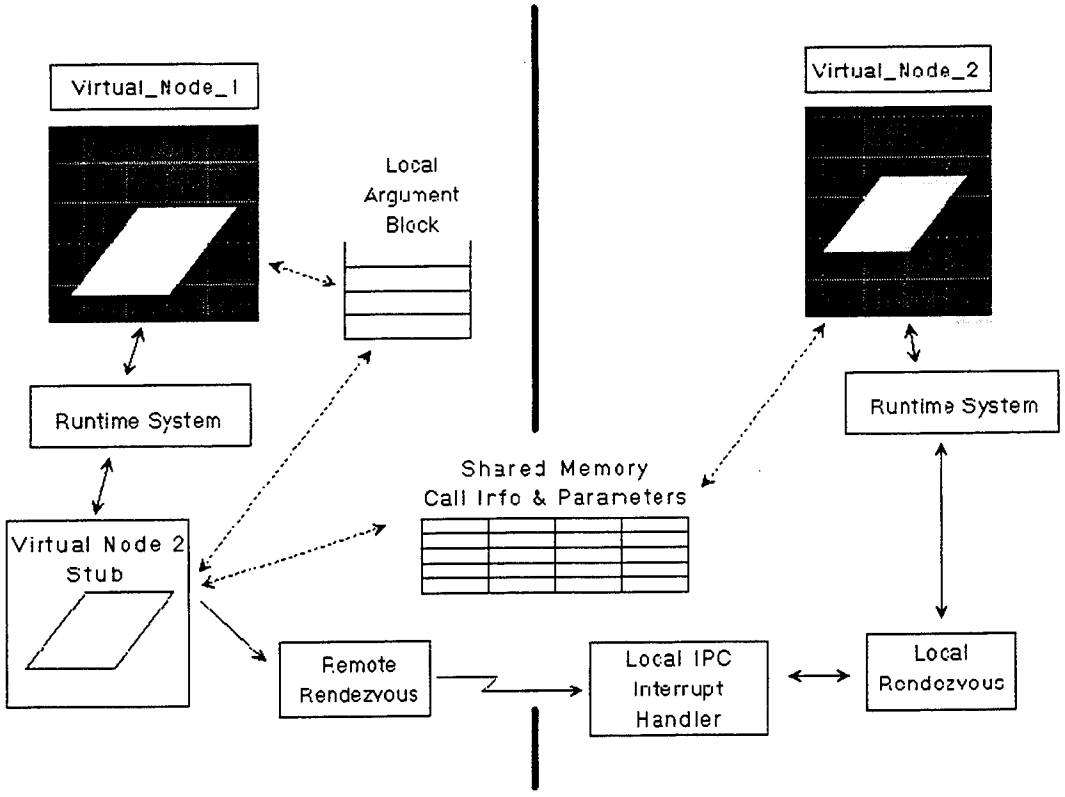

Figure 2. Shared Memory Architecture

5.1.5.1 Shared Memory

Communication among processors within an individual chassis is based upon shared VME memory equally visible to all the processors in that chassis. This memory is identical in behavior to that of the memory local to each processor, with the usual proviso that accesses will be slower due to bus contention. Indivisible machine instructions such as test-and-set or compare-and-swap function properly.

Similarly, global memory is available between the three chassis, and is equally visible to every processor in the target environment (unlike the VME memory). However, indivisible instructions do not function properly, which becomes significant when implementing mutual exclusion mechanisms, discussed later with respect to performance limitations.

5.1.5.2 Stubs

The underlying implementation artifact is the *stub*, as is true of many distributed system designs. In this implementation, however, stubs are tasks. Specifically, stubs take the place of called tasks in the site where calls take place. This approach has the following advantages:

a) The compiler and linker see a complete program since all tasks are accounted for, including those tasks that are actually remote.

b) A pool of stubs is not needed since a one-to-one correspondence of local stubs to remote tasks exists.

c) Each stub can be tailored specifically to the remote task it is representing, so that the extra overhead of a generalized mechanism is not necessary. For example, if a particular stub's entries have no formal parameters, no code is generated to copy the parameters or even to count them in order to determine that no other action is necessary.

The role of each local stub is to handle rendezvous calls from the local site by intercepting the runtime calls which implement the accept statement, recreating the formal argument block in intrachassis (VME) memory or interchassis (Reflective) memory, signalling the request to the remote site, and awaiting the response. For responses indicating success, any returning parameters are then copied back to the local argument block and the caller is resumed via the runtime system calls that implement the end of the accept statement. For responses indicating an exception, the "exceptional" end-of-accept-statement runtime calls are executed with the correct exception indicated to handle propagation to the caller. A portion of a typical stub follows:

```
task body Interface is -- the stub for a task named "Interface"
  ...
begin
    Tasking.Identify_Stub;  -- mark TCB of this task as that of a stub
    ...
    loop
        Tasking.Await_Call( Number_Entries, Entry_Called, Arg_Block, Old_Priority );
        ... copy any mode in or mode in_out parameters to shared memory
        case Tasking.Call_Kind is -- determined from TCB
            when Rendezvous.Unconditional =>
                IntraChassis_Rendezvous.Unconditional_Call( ..., Result );
            ...
        end case;
        if Result = Rendezvous.Completed then
            ... copy any mode out or mode in_out parameters back
            Tasking.End_Accept( Old_Priority );
        else
            Handle_Rendezvous_Error( Result, Old_Priority );
        end if;
    end loop;
end Interface;
```

Early versions of stubs determined the location of the target task each time a call was received, in order to determine whether to make an intrachassis or interchassis rendezvous call. This approach allowed the target task to be on different physical nodes per call, so that transparent fault tolerance could be supported in later versions. Unless fully transparent fault tolerance is supported, however, target tasks will always be on the same remote physical node so the current version of the stubs determines the corresponding target task location only once.

5.1.5.3 Surrogates

On the receiving side, each processor has a single surrogate (interrupt-handler) task that fields all incoming remote rendezvous requests. For each request, the surrogate makes the necessary runtime system call to start the rendezvous with the task being called and signals the caller upon completion. Although this approach results in the serialization of remote rendezvous on a given node, it is not considered unacceptable since a surrogate pool approach would cause the same effect when the pool became exhausted. Since the size of a pool on a given physical node would depend upon the number of virtual nodes mapped interactively to it, the number cannot be determined when building the runtime system extension software. Thus a single surrogate is used for all virtual nodes on any given physical node.

5.1.5.4 Intrachassis Rendezvous

For remote rendezvous requests to a task in the same chassis, the chassis-wide VME memory is used to contain the replicated argument block for both by-copy and by-reference parameters. During elaboration, each task stub obtains a dedicated index into the VME memory areas used for this purpose.

Communication within a chassis is via hardware-supported board-to-board interrupts that transmit a single byte of information to the interrupted processor. These interrupts are part of the standard, commercial multiprocessing support provided by the hardware vendor. In this case the

byte of information transmitted is the index into the VME memory area that contains information about the pending call, such as the entry being called and the caller's arguments.

The local surrogate simply passes the address of the replicated argument block in VME memory directly to the runtime routine that implements the entry call. Thus the argument block is only recreated once: from the local calling processor into the VME memory. Any return parameters are copied back into the local argument block once the rendezvous is completed. (The overhead of this coping will be removed prior to completion of the project.)

5.1.5.5 Interchassis Rendezvous

Rendezvous between chassis are implemented in a very similar manner to calls within a single chassis since the interchassis global "reflective" memory has been made available for distribution support. The major difference involves the lack of the built-in interprocessor message-passing mechanism. Especially at the upper levels, the software is nearly identical since both designs are based upon shared memory.

In place of the interprocessor message-passing facility, reflective memory interrupts are used in combination with dedicated message buffers to provide interchassis communication. Calling stubs simply place the usual information, plus chassis identification, into the incoming call buffer associated with the destination chassis and then generate an interrupt to that chassis. In this case the interrupt is generated via the reflective memory hardware and is always handled first by the processor in the destination chassis that has identification of "processor 1". That is, processor number one in each chassis is the sole interchassis interrupt handler for that chassis. The interrupt handler determines which processor is actually the final destination and then uses the hardware-supported message-passing facility to signal that processor. Thus interchassis communication is in two stages.

Although the reflective memory interrupts could be assigned dedicated interrupt offsets, thereby enabling more direct one-stage communications, only a limited number of destination processors would then be supported since each offset would require a separate VME interrupt priority level. These levels are limited in number and are already partially in use. By having "processor 1" act as the interchassis interrupt multiplexor, no additional limit is imposed on the number of destination processors supported.

Since the global reflective memory in use does not support read-modify-write cycles, the synchronization technique based on test-and-sets in use with VME memory cannot be used. Instead, a pure software approach based on Eisenberg and MacGuire's algorithm [Raynal] for mutual exclusion among N processes has been implemented. (In this case, a "process" is a stub executing on a given chassis.) However, the cost of a software-based mutual exclusion mechanism is obviously high, making rendezvous *between* chassis slower than rendezvous *within* a chassis. Alternative mechanisms with better performance are being developed.

5.1.5.6 Conditional and Timed Entry Calls

Unlike unconditional entry calls, which have straightforward semantics, conditional and timed entry calls require more consideration in a distributed context. Various Ada Interpretations ("AI") have defined the required and/or allowed semantics for distributed calls. AECSS implementation of these calls is in line with these interpretations.

Specifically, conditional entry calls ignore communication delays, as is consistent with RM 9.7.2(4) and with AI-276 (see Appendix).

Timed entry calls are timed from the remote (called) task's site, as this is by far the most simple approach and is not inconsistent with the RM semantics. Timed calls with zero or negative delays are treated as remote conditional calls, as per AI-276.

Since the called site does the timing, the use of timed calls for fault tolerance is not supported. However, such use is conceptually flawed: the RM does not specify distributed failure semantics for timed entry calls. There is no case in which a task in a uniprocessor-based implementation would be unreachable (as opposed to uncallable). In a distributed system this can certainly be the case, but failure semantics are not defined by the language standard.

5.1.6 COTS Runtime System Modifications

The standard COTS runtime system code procured with the cross-compilation system has been modified in order to cooperate with the runtime extensions supporting distribution. As part of the modification, the task control block (TCB) of stub tasks are also altered. These modifications are discussed in the following sections.

5.1.6.1 Code Modifications

Only one module has been altered, that which implements the unconditional, conditional and timed entry calls. Specifically, a check has been added to each call to determine whether or not the call is to a remote, distributed task. If not, then the normal call handling is performed. If the call is to a distributed task then the caller will in fact be dealing with the local *stub* for the remote (called) task. The call must be handled in such a way that the calling task always waits for the call to be completed by the stub's interaction with the remote task. Specifically, the caller must not time-out from a timed call, or return from a conditional call, based on its interaction with the stub. (Unconditional calls are not an issue since the caller waits by definition.) To make the caller wait indefinitely, the flags set at runtime to indicate conditional and timed calls are altered to indicate an unconditional call after the kind of call is noted and time-out values are copied (for timed calls). At the end of the common module implementing all three kinds of call, the augmented code determines the original kind of call and sets the corresponding flags in the proper registers, including success/failure for timed and conditional calls. Thus the compiler-emitted code representing the calling tasks is not aware of the interaction with the extended runtime system, allowing the calling task's code to work as usual.

5.1.6.2 Task Control Block Usage

To determine if the call is to a remote task, that is, that the local task being referenced is actually a stub for the called task, each stub marks its TCB when it first executes. The module described above checks this flag within the TCB. In order to be safely used without interfering with the existing runtime system code, the altered fields within the TCB clearly must not be used by the existing runtime system when their values are modified by the runtime extensions. This effect has been achieved in two ways.

In one case, certain fields within the TCB are never used after activation of the corresponding task. Specifically, each task's TCB has a field which points to the TCB of the activating task. After activation this field is no longer used so it can be altered safely. A count of activating tasks

is also available in the TCB for tasks that activate others. Since stubs never activate tasks this field can also be altered. These fields are used to note timeout values for timed calls and to save the pointer to the TCB of the calling task for reference after the call completes.

In the other case, certain fields are dedicated to optional vendor-supplied functionality that is not used by the project. Specifically, the compiler vendor supports mailbox-based task communications not used by AECSS programs. These fields are used to mark the TCB as that of a stub and to note the kind of call. Use of the mailbox facilities by applications programs should not occur since remote rendezvous is the communication and synchronization mechanism prescribed for application use.

5.1.7 Permanent Restrictions

A few limitations are expected to remain throughout the project. These limitations affect the application designer's ability to use the language in a normal manner, in that a user would not normally have to consider them. All limitations extend from the basic constraint of not altering the compiler and/or linker.

5.1.7.1 Dynamically-Sized Rendezvous Parameters

Although both predefined and user-defined types are supported for remote rendezvous parameters, the size of the actual parameters must be determinable solely from the name of the formal parameter's type (the "type_mark"). Thus, for example, although Ada allows formal parameters to be of an unconstrained array type, such as Standard.String, the current implementation cannot support them since the type_mark doesn't specify the size.

This restriction is a result of the fact that the stubs implement the *semantics* of accept statements for the corresponding interface task's entries without actually performing accept statements (otherwise the rendezvous would be completed *locally*, with the stub, instead of with the intended remote task). The effect of the accept statement is achieved by having the stub call the same runtime routines that the compiler would have emitted in place of the "accept" keyword. Since the accept statement does not literally take place within the stub, the entry's formal parameters cannot be referenced by their names so their sizes cannot be determined at runtime (via an attribute or some other means). Instead the type_marks are used, since they alone are available in the stub. Thus the type_marks must be sufficient to specify the sizes.

5.1.7.2 User-Defined Exception Propagation

The language requires unhandled exceptions within accept statements to become active in both the caller and called tasks (RM 11.5/4). Distributed Ada must therefore support exception propagation across remote rendezvous calls. The same exception must become active in both the local caller and remote called sites.

The language requires exceptions to be uniquely identified (RM 11.1/3). Since an arbitrary number of separate compilation units can declare an arbitrary number of exceptions, it is the case that, prior to linking, an unknown number of exceptions can exist within the program to be created. Thus it is the linker that typically assigns the unique identifiers to all exceptions within a given program. (Note that other techniques are possible; a linker is not mandated by the language. Rather, this description depicts a typical approach that is in fact reflected by the compilation system in use for the 68030 targets.)

The effect of having the linker assign unique identifiers to exceptions is that the runtime system cannot know what exception name corresponds to a given exception number, unless the compiler and linker make this information available by some convention. Since the distributed runtime system must propagate the same exception, that is, the one with the same name, to the remote calling site, the pairing of names and numbers is critical. Because the compiler and linker cannot be modified, the convention supported by the compiler/linker must be used if such exists.

Fortunately the compiler and linker define standard locations at which the numbers of the predefined exceptions can be found. As a result, even though the number assigned to a predefined exception is typically different for each partition (i.e. program), the pairing of name to number can be determined. Consequently only the predefined exceptions can be handled *by name* when handling exceptions propagated from a remote entry call. All other exceptions are anonymous and must therefore be handled via an "others" choice.

5.1.7.3 Select with Terminate Alternative

Since the underlying (hidden) implementation is essentially multiprogramming, statements which require system state information about other parts of the (conceptually single) application program cannot provide the required semantic behavior. For Virtual Nodes under AECSS this restriction is embodied in the select statement with a terminate alternative located within an interface task. However, by definition the only tasks in question are those that are in library unit packages so the issue is inconsequential.

5.1.8 Temporary Restrictions

Some temporary restrictions currently exist. These restrictions are expected to be removed prior to the end of the project. They have been deferred for priority reasons only.

5.1.8.1 Mixed-Size Entry Formal Parameter Order

In order to conserve space, the compiler for the 68030 target "packs" the argument block it constructs for task entry parameters such that later formal parameters, if small enough, are placed in earlier unused contiguous bytes. For example, if a value of type Standard.Character is passed as the first formal parameter, then a value of type Standard.Integer is passed, and then another of type Standard.Character, the third parameter (the character) will be placed in one of the unused bytes prior to the bytes used for the second parameter (the integer). See Figure 3. However, the current argument replication module in the runtime extension software assumes that all parameters are provided their own longword-aligned allocation, which is not what the compiler does. As a result, some parameter declarations will not be handled correctly, and will exhibit unpredictable values when the

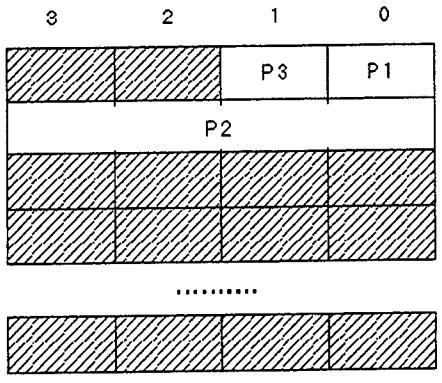

Figure 3. Argument Block Packing

program is executed. For example, the following declaration corresponds to the above example and exhibits the undesired behavior:

```
entry Call( P1 : in Character; P2 : in Integer; P3 : in Character );
```

Therefore application developers must be careful when defining parameters to be passed to remote tasks such that those parameters which occupy less than 4 bytes and do not require longword-alignment come after those that require 4 or more bytes. Work is underway to remove this restriction.

5.1.8.2 Remote Entry Families

Stubs are generated automatically by a program that parses virtual node interface package specifications and produces the corresponding package bodies, with their concomitant interface task bodies. Since the types and data structures that support remote rendezvous are static the total number of entries per interface task must be determinable at stub generation time. Currently, the stub generator does not handle entry families since the entry family declaration syntax includes the use of an identifier to indicate the range of the family. Altering the stub generator to recognize this syntax, and to generate declarations based on attributes of this identifier has been deferred until later in the project.

5.1.9 Configuration Limits

In order to provide maximum predictability all possible data structures are statically allocated. In particular, arrays with constant access times are used instead of linked lists because the performance of linked lists degrade as their lengths increase. As a result, the bounds of these arrays impose limits upon the configuration, and thus the user, in specific areas. These limits are as follows:

There may be at most 50 virtual node interface tasks in a given application program. Because a virtual node may have multiple interface packages containing multiple interface tasks, the number of virtual nodes in an application may be less than the number of interface tasks.

There may be at most 128 pending remote interchassis rendezvous requests per chassis.

There may be at most 128 pending remote interchassis rendezvous acknowledgements per chassis.

The full name of any individual virtual node interface task, which includes the interface package name, may not be longer than 75 characters. For example, the full name of hypothetical task Interface in package Life_Support would be "Life_Support.Interface," which is an acceptable length. As can be seen the limit of 75 characters should be more than sufficient.

There currently exists an upper bound on the size of the rendezvous parameters passed to a task in the same chassis as the caller. The limit is 1024 bytes. This limitation will be removed prior to completion of the project, so that the constraint will be that of physical memory available to the caller's processor. (In particular, the parameters will no longer be copied to VME memory in order to enhance performance.)

The parameters passed to a task in a remote chassis from that of the caller are limited to a maximum of 1024 bytes. This value is configurable, but unlike rendezvous within a given chassis, an upper bound will always exist. Frequent remote rendezvous which are required to transmit large amounts of data are not anticipated.

Note that these limits are generally arbitrary and are alterable by the systems configuration architect. However, physical memory constraints must be accommodated when changing the upper bounds. This is especially the case with the global reflective memory shared among the three chassis, since it is considerably smaller than the VME (DRAM) memory dedicated to each chassis.

5.1.10 Portability Issues

For good software engineering reasons as well as the fact that more than one instruction set architecture (ISA) was originally to be supported, the code has been written with portability in mind. Two aspects of portability are of importance: implementation-*defined* facilities, and implementation-*dependent* facilities. Each is discussed below.

5.1.10.1 Implementation-Dependent Facilities

Implementation-dependent facilities have been used extensively, such as address clauses and machine code inserts for low-level synchronization mechanisms, but they have been hidden as much as possible. For example, the package which allows the runtime system extensions to determine which processor is currently executing is implemented in such a way as to hide the fact that it references the static on-board memory of the processor executing the code. The package is shown below.

```
package Processor is

    type Identifier is range 1 .. 21;

    function This_CPU return Identifier;
    ...
end Processor;

with System;
package body Processor is

    VME_Slot_Number : System.Unsigned_Byte;
    for VME_Slot_Number use at System.To_Address( 16#FFC0_0801# );

    function This_CPU return Identifier is
    begin
        return Identifier( VME_Slot_Number );
    end This_CPU;
    ...
end Processor;
```

5.1.10.2 Implementation-Defined Facilities

Use of implementation-defined facilities has been kept to a minimum. However these mechanisms are sometimes the best or even the *only* way to achieve some goal so they have been used where appropriate. For example, the compiler supports an implementation-defined attribute that provides runtime type class information. Specifically, one can determine if a given type is an array type, a record type, a scalar type, and so forth. This attribute is used to determine if the compiler is passing a given actual parameter by reference or by copy.

```
case TypeMark'Type_Class is
    when Type_Class_Array | Type_Class_Record =>
        -- handle by-reference parameter
    when others =>
        -- handle by-copy parameters
end case;
```

At present, composite objects are always passed by reference. Although it could change in a future release of the compiler, such a change is not expected. In any case only one module would be affected.

Another case concerns the occasional instance in which a specific parameter-passing mechanism is necessary to ensure proper behavior. An example is a mechanism for providing a mutual exclusion mechanism in a multiprocessing environment, in which the locking object passed to the lock and unlock routines must be passed by reference so that multiple concurrent callers do not operate on *copies* of the locking mechanism. This effect can be achieved through fairly inelegant techniques in which the effect of pass-by-reference is achieved. However, the implementation supports a pragma to tell the compiler how to pass parameters, which results in the simple and clean interface shown below.

```
package MultiProcessor_Mutex is

    type Spin_Lock is private;

    procedure Lock( The_Lock : in out Spin_Lock );
    procedure Unlock( The_Lock : in out Spin_Lock );

    pragma Call_Sequence_Procedure( Lock,  Mechanism => Reference );
    pragma Call_Sequence_Procedure( Unlock,  Mechanism => Reference );

private
    ...
end MultiProcessor_Mutex;
```

5.1.11 Host Tool Suite

SBS personnel have implemented various host-based tools to automate the production and execution of distributed software. Specifically, a stub generator has been written to automatically create tailored stubs for interface tasks, and a node allocator has been written to allow the user to interactively allocate virtual nodes to physical nodes.

5.1.11.1 Stub Generator

The Stub Generator parses interface package specifications containing singleton interface task specifications and produces the corresponding package body with its task body stubs. As a result the user need only address the development of the application; system software generation is handled automatically upon demand. The resulting stub is tailored for the specific characteristics of the task it represents. For example, if no parameters are to be passed to any entries the formal parameter-handling code is not generated at all. Similarly if only one entry exists no code is generated to determine which entry has been called.

5.1.11.2 Node Allocator

In order to provide maximum flexibility, the allocations of virtual nodes to physical nodes is not fixed. In particular it is not fixed by the application source code via mechanisms such as pragmas or comments, as that approach would limit the flexibility of the developed applications. Instead a "map" is used in the runtime extension software, which specifies the locations of virtual nodes within a single chassis and within the "network" of chassis. Thus, for example, when a stub is to determine whether to use InterChassis_Rendezvous or IntraChassis_Rendezvous it checks the map to see if the called task is in the same chassis. Similarly, the units within IntraChassis_Rendezvous check the map to determine which processor within the chassis contains the called task.

The map is built prior to runtime and then compiled as an individual package body. The body is generated and compiled automatically, based upon information supplied by the user in an interactive mapping session.

5.2 Fault Tolerance

Two major classifications of fault tolerance may be observed: transparent and application-level. The AECSS project uses one form in a configurable capability described below.

5.2.1 Transparent Fault Tolerance

A system that supports true transparent fault tolerance does so in such a way that the application is largely unaware of failures that occur. Thus recovery, in the form of replacement of software units executing on failed processors, is handled without application intervention or direction. For so-called "passive" software units, which are dormant until called and do not have their own state, this is not particularly difficult since an arbitrary number of replicated units can be deployed prior to execution (or at runtime) and switched among as necessary. A passive unit would be represented in Ada, for example, as a package containing a data structure, with subprograms that alter that state when called. Essentially these are "cold stand-bys".

However, "active" units, represented in Ada as tasks, pose a significant difficulty for transparent fault tolerance support. The difficulty results from the fact that task objects have their own unique, individual state and their own thread of control. Internal operations by the thread can alter the state, as can external entry calls. In order for the fault tolerance support to be transparent, replicated tasks must maintain consistent internal states with the primary task, which means that each replicant must at least achieve the effects seen by those rendezvous occurring with the primary task, i.e., they must be "hot stand-bys". The degree of difficulty imposed upon the underlying implementation in such a system is severe.

5.2.2 Application-Level Fault Tolerance

In contrast to transparent fault tolerance, the application can be made responsible for some or all aspects of recovery. In such an implementation, the underlying system software detects a failure and then notifies the application. The application is then responsible for determining what response is necessary or appropriate. Obviously this approach is considerably more simple than the transparent implementation. It is also the case that application involvement is sometimes the only reasonable approach, since many failures are application-specific.

Several issues must be considered in the design and implementation of application level fault tolerance systems. For example, should all units be notified of a given processor's failure or just a subset; should the processors' health be checked actively or only when communications are requested; how many active checkers should exist in order to address the single point of failure issue, and so on. Each decision has an impact upon application responsiveness and performance characteristics.

5.2.3 AECSS Fault Tolerance Approach

Because support for distribution was the primary goal of the project, and because predictable and acceptable performance was desired, application-level fault tolerance support was chosen for the project. Support for fault tolerance can be included or omitted as appropriate, per application requirements, with the following characteristics:

Active checking is performed for the sake of timely detection and notification. All processors check all other processors in use, in order to avoid a single point of failure.

The interface to the application is in the form of the predefined exception TASKING_ERROR, which is propagated to the calling task whenever the called task is on a processor that has failed either *before* or *after* the call is made.

Currently, only rendezvous occurring within a given chassis support fault tolerance. Support for rendezvous between chassis is under development.

5.3 Deterministic Scheduling

Part of the research effort was scheduled to include an examination of the applicability of schedulability theory to distributed targets. Implementing the distributed system software has so far preempted the effort.

5.4 Deterministic Storage Utilization

Because the source for the compiler was not acquired and since the compiler must be modified in order to implement deterministic storage unitization, effort was limited to determining desirable behavior and potential user-level interfaces. Both are best described in [SofTech].

6 Lessons Learned

AECSS is an on-going project. As such, the lessons learned so far pertain mainly to the development of distributed Ada, as discussed in the preceding sections. As the project progresses into the next phase, in which the virtual node technology is applied to an avionics application, different lessons will no doubt be learned.

6.1 Logistics

Some of the lessons learned have to do with the logistics of managing a project. Specifically, the issues involved with supporting nonstandard products are discussed in these sections.

6.1.1 Retargeting COTS Compilers

The compiler chosen did not at the time support the FORCE brand of 68030 processors. SBS personnel retargeted the kernel to the FORCE boards with little difficulty, except for clock functionality. An error in the clock setup was discovered and corrected much later in the project without significant impact.

The main issue, then, is the fact that project personnel are responsible for tracking any changes made by the compiler vendor that might affect the code altered by SBS. To date there have been no problems, but as a general practice use of nonstandard products is not recommended.

6.1.2 Ethernet Facilities

SBS personnel chose to use an Ethernet product for the applications software that did not have an Ada interface (none did at the time). As part of the project, SBS personnel wrote an Ada driver to execute with the application code in order to make the Ethernet communications facility available to users. Vendor documentation was adequate but only from the end-user's point of view. Since SBS personnel were writing the driver rather than applications, the documentation was only indirectly helpful. The only model or documentation for the driver was in fact the code of the driver executing on the Ethernet board itself; the application's driver is a mirror image of it and does symmetric operations. Unfortunately, the board's driver was written in the 'C' language by the president of the vendor company. He had long since forgotten the "how" and especially the "why" of the 'C' implementation choices, some of which were unintelligible. Creation of the application-side driver was correspondingly difficult.

The resulting Ada driver, which supports both TCP/IP and UDP "datagrams," works well in spite of the difficulties in creation. However, the production of the final version would likely not be cost-effective, assuming an appropriate product is available. (Again, such was not the case.)

6.2 Research

Research efforts primarily focused upon the implementation of Distributed Ada. The areas of Deterministic Scheduling and Deterministic Storage Management were also planned for exploration. These three research topics are discussed in the following sections.

6.2.1 Distributed Ada

The implementation of Distributed Ada went well, especially due to judicious early choices concerning the chosen granularity of partitioning and to the availability of shared memory (both within and between chassis). Shared memory is becoming an increasingly viable alternative to LAN technology, and thus does not reduce the applicability of the resulting implementation.

6.2.1.1 Virtual Nodes

The Virtual Nodes concept significantly eased implementation of Distributed Ada, due primarily to the removal of required support for remote allocation and reference of arbitrary objects. For example, variables within expressions would otherwise have been able to be located on processors remote from each other. As part of the Virtual Node Design Method, shared variables are precluded from being visible to remote references, as are any other entity except singleton tasks. Since the runtime system already supports a dynamic representation of tasks, and since only the runtime system software was modified and extended, remote task entry calls were relatively easy to support. Other forms of distribution support, such as remote procedure calls, would have required *compiler* support. The choice of applying Virtual Nodes has been central to the successful implementation of Distributed Ada.

6.2.1.2 Shared Memory

Furthermore, the decision to use the global memory shared among chassis significantly accelerated the implementation, since the shared-memory approach was already implemented for rendezvous within any individual chassis, i.e., using VME memory shared among the processors. Differences between the memory shared among the processors in a given chassis and the memory shared among all chassis required some modifications to the distribution support routines, but the design was unaltered.

6.2.1.3 Performance Limitations

Certain aspects of the implementation's initial constraints lead to inherent performance limitations. In particular, use of a COTS compilation system without modifications, and use of shared memory without indivisible instruction support have resulted in acceptable, but less than optimal, performance. These two issues are discussed in the following sections.

6.2.1.3.1 Unmodified COTS Compilation System

Since the compiler and linker are not modifiable, the stubs for interface tasks are tasks, rather than procedures. Stubs are in fact the local executable's version of the interface task bodies, making the linker believe that all tasks are present, including those that are actually remote. This approach satisfies all the rules pertaining to program unit closure within the program library but requires calling tasks to perform the same number of tasking context switches as would be the case in a normal, nondistributed rendezvous. As a result, performance measurements will always be a little slower than those of implementations in which the compiler has been modified to have the stub interactions be mere subprogram calls.

6.2.1.3.2 Memory Without Indivisible Instruction Support

Because the global memory shared among the several chassis does not support indivisible instructions such as test-and-set or compare-and-swap, mutual exclusion mechanisms which work on the VME memory within a chassis cannot work on the global memory shared between chassis. As a result, alternative methods of achieving mutual exclusion are necessary, and are likely to be slower than the very simple operations based on indivisible machine instructions. The current approach is implemented entirely in software, without support from the underlying machine, and is a major factor in the significant difference between intra- and inter-chassis rendezvous performance.

6.2.1.4 Functional Limitations

Since the compiler and linker are not modifiable, certain language functionality is not currently supported. Specifically, user-defined exceptions propagated from remote accept bodies during rendezvous cannot be handled by their names (as explained earlier). They can, however, be handled via the "when others" clause, but this is a deviation from the predefined semantics of the language. Likewise, unconstrained array parameters passed to remote tasks are currently not supported since the array component type may not be declared lexically within the interface package specification. As a result, the component type's size is unknown, and since the dimensionality is similarly unknown to the stub, the entire array parameter's size is unknown. One approach under consideration is to limit support to array types declared within the interface package specification.

6.2.2 Deterministic Scheduling

Although much has been written about schedulability analysis for local area networks, little has been done with the kind of shared memory connecting the various chassis. However, research into this area requires considerably more time and resources than originally expected.

6.2.3 Deterministic Storage Utilization

Because the compiler was not modifiable, deterministic storage utilization was not implementable.

7 Interim Conclusions

Implementation of Distributed Ada can be accomplished without special-purpose operating systems or compilation systems. Modification to the COTS runtime system was minimal in the extreme: one module was augmented. Furthermore, the runtime extensions and tools are less than 3,000 lines of Ada code (measured in terms of "meaningful" semicolons). As a result, the potential for cost-effective support of distributed, advanced embedded avionics Ada applications has been clearly demonstrated.

8 References

Atkinson C. et al. (1988). Ada For Distributed Systems, Cambridge University Press.

Cornhill, D. (1984). "Four Approaches To Partitioning Ada Programs For Execution on Distributed Targets," *Proceedings of the IEEE Conference on Ada Applications and Environments*, pp. 153-164.

Eisenhauer, G. et al. (1986). "Distributed Ada: Methodology, Notation and Tools," First International Conference On Ada Programming Language Applications For the NASA Space Station, pp. B.3.2.1 - B.3.2.8.

Intermetrics (1991). "Draft Ada9X Mapping Document, Volume II, Mapping Specification," Ada9X Project Report, December 1991.

Knight, J. (1984). "On The Implementation and Use of Ada On Fault-Tolerant Distributed Systems," Ada Letters, 4(3), pp. 53-64.

Raynal, M. (1986). Algorithms For Mutual Exclusion, MIT-Press.

Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983, American National Standards Institute, 1983.

Rogers, P., McKay, C. (1986). "Distributed Program Entities In Ada," First International Conference On Ada Programming Language Applications For the NASA Space Station, pp. B.3.4.1 - B.3.4.13.

SofTech, 1988. *Storage Management In Ada: Three Reports*, SofTech Document WO-123.

Volz, R.A. et al. (1985). "Some Problems In Distributing Real-Time Ada Programs Across Machines," *Ada In Use, Proceedings of the Ada International Conference*, pp. 72-84.

Wellings, A., Tomlinson, G., Keefe, D., Wand, I. (1984). "Communications Between Ada Programs," *Proceedings of the IEEE Conference on Ada Applications and Environments*, October 1984, pp. 145-152.

Appendix A "AI-276"

The following is the text of AI-276.

87-02-23 AI-00276/07 1

| !standard 09.07.03 (04) 87-02-23 AI-00276/07
| !standard 09.07.02 (01)
| !class ramification 85-02-27
| !status approved by WG9/AJPO 87-02-20
| !status approved by Director, AJPO 87-02-20
| !status approved by Ada Board (21-0-2) 87-02-19
!status approved by WG9 85-11-18
!status committee-approved (7-1-2) 85-02-27
!status work-item 85-02-04
!status received 84-08-27
!references 83-00406, 83-00437
!topic Rendezvous that are "immediately possible" vs. timed entry calls

!summary 85-09-14

A timed entry call with a zero or negative delay issues an entry call that is canceled only if a rendezvous is not immediately possible. RM 9.7.2(4) specifies the conditions under which an entry call is immediately possible. In a distributed implementation of Ada, it may take a non-negligible amount of time to determine whether an entry call is "immediately" possible.

!question 85-10-29

RM 9.7.2(1) states, "A conditional entry call issues an entry call that is then canceled if a rendezvous is not immediately possible." Paragraph 4 of the same section then goes on to state the conditions under which such an entry call must be canceled. It is not clear if paragraph 4 is meant to provide a complete definition for the word immediate as used in paragraph 1, or merely to list a necessary set of conditions under which such an entry call must be canceled. While these two statements do not appear inconsistent in the uniprocessor case, the exact interpretation is much less clear when considering the impact on distributed systems.

In distributed environments several interpretations are possible. One interpretation is that due to non-negligible inter-nodal communication delays there can be no "immediate" acceptance of distributed entry calls; hence, non-local conditional entry calls are always canceled. A less restrictive interpretation is to say the determination of immediacy uses the criteria stated in 9.7.2(4) (i.e., the determination of immediacy is completely independent of any communication delays that may be present in the implementation of an arbitrary distributed system). For applications where the absolute elapsed time between a call and accept is important, timed entry calls should be used.

Which interpretation is correct?

!response 85-09-14

Since RM 9.7.2(1) says an entry call (for a conditional entry call) is canceled "if a rendezvous is not immediately possible," and RM 9.7.2(4) specifies the conditions under which an entry call is canceled, RM 9.7.2(4) implicitly defines when an entry call is "not immediately possible". Determining that a called task is able (and willing) to accept a call might take a long time, especially in a distributed processing environment. Nonetheless, since the definition in 9.7.2(4) does not mention time, "immediately possible" is to be understood as specifying requirements on the state of the called task (and its willingness to accept the call in the case of a selective wait) rather than a requirement that the call be accepted within some small interval of time.

RM 9.7.3(3) says, for a timed entry call:

> If a rendezvous can be started within the specified duration (or immediately, as for a conditional entry call, for a negative or zero delay), it is performed ... Otherwise the entry call is canceled when the specified duration has expired.

This means that if the specified delay is exactly zero or negative, execution of the timed entry call takes as long as needed to decide whether the call can be accepted "immediately," in the sense of 9.7.2(4). If the delay is nonzero and positive, the entry call can be canceled as soon as the delay expires (if it has not already been accepted). If the delay is small enough, it even might be canceled before it has been possible to decide if the call can be accepted "immediately".

Acronym List

AECSS	Ada Embedded Computer Software Support
ANSI	American National Standards Institute
AI	Ada Interpretation
COTS	Commercial Off The Shelf
DIADEM	DIstributed Ada DEMonstrated
ISA	Instruction Set Architecture
ISO	International Standards Organization
LAN	Local Area Network
RM	Reference Manual for the Ada Programming Language
RPC	Remote Procedure Call
RTS	Runtime System
RTSE	Runtime Support Environment
TCB	Task Control Block
VME	VersaModule Eurocard
VN	Virtual Node
YDA	York Distributed Ada